DOCKET NO: P3355US1

(119-0035US)

APPLICATION

FOR

UNITED STATES LETTERS PATENT

TITLE: Improved Blur Computation Algorithm

INVENTOR: Mark Zimmer

Express Mail No:
EV 405193286 US
Date:
4/16/04

Prepared by: WONG, CABELLO, LUTSCH, RUTHERFORD & BRUCCULERI, L.L.P.
HOUSTON, TEXAS

IMPROVED BLUR COMPUTATION ALGORITHM

Background

[0001] The present invention is in the field of computer graphics processing. Particularly the present invention relates to techniques for increasing the efficiency of image-processing related computations. The invention described herein is particularly applicable to use in systems having a central processing unit (CPU) operating together with a dedicated graphics processing unit (GPU). Various implementations of such an architecture are described in assignee's co-pending patent applications: "System for Reducing the Number of Programs Necessary to Render an Image," by John Harper, serial no. _____; "System for Optimizing Graphics for Operations," by John Harper, Ralph Brunner, Peter Graffagnino, and Mark Zimmer, serial no. _____; "System for Emulating Graphics Operations," by John Harper, serial no. _____; and "High Level Program Interface for Graphics Operations," by John Harper, Ralph Brunner, Peter Graffagnino, and Mark Zimmer, serial no. _____, each filed concurrently herewith and incorporated herein by reference in its entirety. Although the methods and techniques described herein are particularly applicable to systems having a single CPU/single GPU architecture, there is no intent to restrict the invention to such systems. It is believed that the methods and techniques described herein may be advantageously applied in a variety of architectures.

[0002] In the object-oriented programming context of most modern graphics processing systems, there are generally four types of objects available to a programmer: images, filters, contexts, and vectors. An image is generally either the two dimensional result of rendering (a pixel image) or a representation of the same. A filter is generally high-level functions that are used to affect images. A context is a space, such as a defined place in memory where the result of a filtering operation resides. A vector is a collection of floating point numbers, for example, the four dimensional vector used to describe the appearance of a pixel (red, blue, green and

transparency levels). Each of these definitions is somewhat exemplary in nature, and the foregoing definitions should not be considered exclusive or otherwise overly restrictive.

[0003] Most relevant to the purposes of the present invention are images and filters. In an embodiment of the present invention, filter-based image manipulation may be used in which the manipulation occurs on a programmable GPU. A relatively common filter applied to images is a blur. Various blurs exist and are used for shadow, the depiction of cinematic motion, defocusing, sharpening, rendering clean line art, detecting edges, and many professional photographic effects. A special blur is the Gaussian blur, which is a radially symmetric blur. Other, more complicated blurs and other convolution operations can often be separated into linear combinations of Gaussian blurs. Because the Gaussian blur is the cornerstone of many image processing algorithms, it is essential to have a fast way of computing it. It is even more desirable to have a way of computing a Gaussian blur that does not tie up the CPU in the calculation.

[0004] Modern programmable graphics processing units (GPUs) have reached a high level of programmability. GPU programs, called fragment programs, allow the programmer to directly compute an image by specifying the program that computes a single pixel of that image. This program is run in parallel by the GPU to produce the result image. To exactly compute a single pixel of Gaussian blur with any given radius it is technically necessary to apply a convolution over the entire source image. This is far too computationally intensive to implement. In practice, only approximations are calculated. To compute the approximation, it is important to use a minimum number of source image lookups (texture lookups). GPU fragment programs typically only allow a small maximum number of textures. Thus a scheme which minimizes the number of passes and maximizes the blurring work done with each pass is sought.

Summary

[0005] The present invention relates to an improved blur computation algorithm. The proposed algorithm accomplishes a blur of an image using fragment programs on a GPU. Alternatively, the blur may be computed on a CPU through emulation or directly programmed. Modifications of the program are possible that accomplish motion blur, zoom blur, radial blur, and various other forms of blur that vary across an image computed for the purpose of simulating depth-of-field.

[0006] Hierarchical blur in fragment programs on GPUs is used to compute Gaussian blurs of images. Hierarchy means different blurs, spaced more or less evenly in the logarithm of the radius. The blur algorithm may be modified to do more or fewer lookups per step to achieve greater radius-varying continuity, resulting in levels and sublevels of hierarchy. The blur algorithm also features linear interpolation between hierarchical levels and sublevels to achieve even greater radius-varying continuity. Additional advantages of the blur algorithm include conceptually one-dimensional methods to achieve motion blur, zoom blur, and radial blur. Also described are functions that spatially interpolate between hierarchical levels and sublevels to achieve blurs that vary their character across an image.

[0007] In the object-oriented programming context of the present invention, there are generally four types of objects available to a programmer: images, filters, contexts, and vectors. An image is generally either the two dimensional result of rendering (a pixel image) or a representation of the same. A filter is generally high-level functions that are used to affect images. A context is a space, such as a defined place in memory where the result of a filtering operation resides. A vector is a collection of floating point numbers, for example, the four dimensional vector used to describe the appearance of a pixel (red, blue, green and transparency levels). Each of these definitions is somewhat exemplary in nature, and the foregoing definitions should not be considered exclusive or otherwise overly restrictive.

[0008] Most relevant to the purposes of the present invention are images and filters. In an embodiment of the present invention, filter-based image manipulation may be used in which the manipulation occurs on a programmable GPU.

Brief Description of the Drawings

[0009] Figures 1A–E illustrate a plurality of functions employed in a digital convolution filter for computing a blur in accordance with the present invention.

[0010] Figure 2 illustrates diagrammatically an input image of a blur filter computed in accordance with the present invention.

[0011] Figure 3 illustrates diagrammatically and output image of a blur filter computed in accordance with the present invention.

[0012] Figure 4 illustrates a spaced function, which diagrammatically represents a second order primary kernel employed in a digital convolution filter for computing a blur in accordance with the present invention.

[0013] Figure 5 illustrates a spaced function, which diagrammatically represents a third order primary kernel employed in a digital convolution filter for computing a blur in accordance with the present invention.

[0014] Figure 6 illustrates the relationship between a source image, an image obtained by applying a half convolution step, and an image obtained by applying two half convolution steps or a full convolution step.

[0015] Figure 7 is a flowchart illustrating a blur process in accordance with the present invention.

[0016] Figure 8 is a tree diagram illustrating a Gaussian blur computed using a process in accordance with the present invention.

Detailed Description

[0017] An improved blur computation algorithm is described herein. The following embodiments of the invention, described in terms applications compatible with

computer systems manufactured by Apple Computer, Inc. of Cupertino, California, are illustrative only and should not be considered limiting in any respect.

[0018] A blur is a filter that may be applied to an image. At its most basic level, a blur is simply a convolution. Referring to Fig. 1A, a basic blur function is illustrated, which is the smallest box blur. This is also known as a minimum blur. Convolved with itself, this yields the blur illustrated in Fig. 1B. The function may be normalized (i.e., reduced to unit gain), and the normalized version of the blur function of Fig. 1B is illustrated in Fig. 1C. If this resultant function is convolved with itself, the blur function illustrated in Fig. 1D results. The functions may be normalized (i.e., reduced to unit gain), and the normalized version of the blur function of Fig. 1D is illustrated in Fig. 1E. As can be seen from Figs. 1A–1E, each subsequent convolution operation increases the width (and thus the standard deviation) of the function.

[0019] As an example, suppose that the blur of Fig. 1E were to be applied to the image illustrated schematically in Fig. 2, with the result being the image illustrated schematically in Fig. 3. Note that this example illustrates only a horizontal pass. The images in Figs. 2 and 3 are made up of pixels arranged into n rows and m columns. Each pixel has a value, which may, for example, be a vector value that specifies color (in red, green, and blue components) and transparency. Each pixel in the output image (Fig. 3) is computed as a weighted average of the values of the corresponding pixel and a predetermined number of surrounding pixels in the input image (Fig. 2). For example, pixel X has a value of 1/16*A+1/4*B+3/8*C+1/4*D+1/16*E. For reasons of computational efficiency, described in greater detail in the incorporated references, it is advantageous to create a GPU fragment program corresponding to each of the various convolutions. Thus a GPU fragment program might be created to perform the convolution of Fig. 1B, and a separate GPU fragment program might be created to perform the convolution of Fig. 1D.

[0020] As described in more detail below, by multiple applications of the repeated convolution described with reference to Figs. 1A–1E, it is possible to obtain a blur that approximates a Gaussian blur. A Gaussian blur is radially symmetric, and is

dimensionally separable. Thus the blur may be performed first in the horizontal direction with the vertical blur being performed on the result or vice versa.

[0021] When specifying a Gaussian blur, the key parameter is the blur radius. The blur radius is the standard deviation of the Gaussian distribution convolved with the image to produce the blur. As can be seen from Figs. 1A-1E, the higher standard deviation functions (i.e., higher blur radii) require a greater number of samples to compute a particular pixel value. For example, the blur of Fig. 1B requires three samples from the source image, while the blur of Fig. 1D requires five samples. Thus creating ever higher ordered convolution functions becomes so computationally intensive as to be extravagant and inefficient. Additionally, it requires the creation of an undue multiplicity of fragment programs to account for the higher blur radii.

[0022] However, the inventor has discovered that an improved blur may be computed using the function of Fig. 1D as a basis. For purposes of the following description, the coefficients of Fig. 1D are referenced, although in a practical implementation, the normalized coefficients of Fig. 1E are used. A GPU fragment program is preferably used to implement the convolution of this function, and this fragment program will be referred to as first kernel or K1. The inventor has discovered blurs for higher radii may be computed using the first kernel with different coefficient spacing, for example as illustrated in Figs. 4 and 5. These expanded functions have larger variances (and thus larger standard deviations and blur radii), yet still require only five samples from the original image. This implementation advantageously allows the same fragment program to be used, the only change is the loop variable (sample spacing), which increases with each step in accordance with 2^{n-1}. Thus the second order kernel (K2) is illustrated in Fig. 4 (sample spacing is $2^{2-1}=2$). Similarly the third order kernel (K3) is illustrated in Fig. 5 (spacing is $2^{3-1}=4$).

[0023] The variance of the original image, unconvolved, is 1/12. The variance of K1 (the normalized version) is 1/3. As is well known from basic probability theory, the variance of a distribution is equal to the square of the standard deviation of the distribution, which is also the blur radius. K2 has a variance of 4/3, and K3 has a

variance of 16/3, thus corresponding to higher blur radii. However, it will be appreciated that with each step, the blur radius (standard deviation) doubles. Thus the variance quadruples with each step.

[0024] The variance of the first intermediate result after applying K1 to the original (unconvolved) image is 1/12 + 1/3. The variance of the second intermediate result after applying K2 to the first intermediate result is 1/12 + 1/3 + 4/3, and so forth. The actual blur radius (standard deviation) of the first intermediate result is sqrt(1/12 + 1/3) = approximately 0.6455. The actual blur radius (standard deviation) of the second intermediate result is sqrt(1/12 + 1/3 + 4/3) = approximately 1.3229.

[0025] Unfortunately, doubling the blur radius with each step leads to an undesirable "popping" between increasing blur radii, i.e., with each step increase in blur radius the algorithm must double the blur radius. Thus the inventor has discovered that half steps may be computed for blur radii between the primary steps (kernels) and the result may be interpolated to approximate a blur for the desired radius.

[0026] Thus, assuming that a user has selected a desired blur radius R_d, $f(R_d)$ gives the number of convolution passes required and the number of interpolations required. The source code to determine the number of passes and interpolations required is included as Appendix A. Interpolation is required because often the desired radius R_d will lie between two different results of a full convolution. The interpolation fraction specifies where between the two results so that the results may be interpolated to get the desired R_d.

[0027] The interpolation technique is illustrated more generally in Fig. 6. A beginning image A is illustrated. A full step, a doubling of the blur radius, will produce the image A". A half step, equivalent to multiplying the blur radius by $\sqrt{2}$ (approx 1.4) will produce the image A'. A second half step performed on the image A' will also produce the image A". Assuming the desired blur radius lies between image A and image A', only the first half step is computed, and the interpolation is performed. If the desired result lies between image A' and image A", it is necessary to compute the first half step (image A') and the second half step (image A") and the desired result is

obtained by interpolation of these two results. This process is also generally illustrated by the flowchart in Fig. 7.

[0028] The half step function, which is also preferably implemented in its own GPU fragment program known as a secondary kernel (S1) performs a convolution of the image with the function illustrated in Fig. 1B (although normalized). For half steps of higher orders, the same secondary kernel is used, but with coefficient spacing increasing on the same order and corresponding to the primary kernel, i.e., 2^{n-1}.

[0029] So, referring back to the flow chart in Fig. 7, the desired variance V_d is computed as the square of the desired blur radius Rd. The program flow enters decision box 701. At this point it is determined whether the desired variance required is greater than the cumulated variance after completion of the next step. If it is, meaning that more than one full step will be required, the next full pass is computed at step 702. The flow then returns to decision box 701. If the desired variance required is not greater than the cumulated variance after completion of the next step, control passes to decision box 703.

[0030] At decision box 703, it is determined whether the desired variance is greater than the cumulated variance after completion of a half pass. If it is not, a single half pass is computed (step 704), and the results of the half pass are interpolated with the prior result (step 705). If the desired variance is greater than the cumulated variance after completion of a single half pass, it is necessary to compute two half passes (steps 706 and 704). The result is then determined by interpolation (step 705) between the results of the two half passes. Although the result of the two half passes is the same as the result of a full pass, it is computationally more efficient to compute the second half pass rather than an additional full pass. This is because the second half pass is performed on the result of the first half pass, while the full pass would require the same source image as the first half pass. Thus fewer memory read/writes are required to perform two half passes than to perform a half pass and a full pass. Similarly, the fragment program to compute the half pass is already available, whereas

a memory read/write would be necessary to retrieve the primary kernel to compute a full pass.

[0031] As described, only a single interpolation step is described. However, if desired, greater levels of radius-varying continuity could be achieved by achieving additional levels of interpolation, for example using a tertiary step or kernel using a lower order convolution. Additionally, the algorithm may be modified by one of ordinary skill in the art to perform more or fewer lookups per step to achieve greater radius-varying continuity, resulting in levels and sublevels of hierarchy.

[0032] As noted, the Gaussian blur is radially symmetric and axis-separable, thus each of these steps is performed for each axis of the image. A complete process diagram of a Gaussian blur is illustrated in Fig. 8. As can be seen from Fig. 8, the specified blur radius required more than two complete passes and less than three complete passes. Thus the primary kernel is executed on the original image in the horizontal direction. The primary kernel (K1) is then executed on the intermediate result in the vertical direction to complete the first pass. The second pass comprises execution of the primary kernel having a coefficient spacing of two (K2) in the horizontal direction, with the same kernel executed in the vertical direction on the intermediate result to complete the second pass.

[0033] Assuming the blur radius is less than a half step more than two full passes, the right hand branch is followed, wherein the secondary kernel is applied horizontally to the result of the second pass to produce an intermediate result. The secondary kernel is then applied vertically to this intermediate result to produce the result of the complete half pass. The final image is then determined by interpolation between the result of the second pass and the result of the complete half pass. If the desired blur radius is more than a half step more than two full passes, the left hand branch is followed. In this branch, two half steps are computed, with the final result again determined by interpolation.

[0034] Additionally, the algorithm may be modified to achieve blur effects other than a Gaussian blur. It may also be used advantageously to achieve conceptually one-

dimensional blurs such as motion blur, zoom blur, radial blur, spiral blur, and others. For these blurs, a full pass consists of a single kernel rather than a horizontal and a vertical kernel.

[0035] For motion blur, the sample-to-sample spacing is a two-dimensional subpixel-accurate displacement representing an angular offset from the destination sample position. The higher-order kernels use this spacing times 2n-1.

[0036] For zoom blur, the sample locations are computed in a different manner. First a vector v is computed, which is a two-dimensional subpixel-accurate displacement representing the specific offset from the center of the zoom blur to the destination sample position. A fraction f, typically close to but not equal to 1 (e.g. 0.98), is used to compute the sample positions. The sample positions are at locations equal to the center of the zoom blur plus v times the following five factors: f-2, f-1, 1, f, f2. The fraction f can be changed to compute more or less accurate zoom blurs.

[0037] For radial blur, the sample locations are computed in another manner. First a vector v is computed, which is a two-dimensional subpixel-accurate displacement representing the specific offset from the center of the zoom blur to the destination sample position. An angle a, typically close to 0 degrees (e.g. 0.25 degrees), is used to compute the sample positions. The sample positions are at locations equal to the center of the zoom blur plus v rotated by the following angles: -2*a, -a, 0, a, 2*a. The angle a can be changed to compute more or less accurate radial blurs.

[0038] For spiral blurs, the sample locations are computed using a composition of the multiplication used in zoom blur and the angular rotation used in radial blur.

[0039] Gaussian blur may also modified to produce sharpening, highpass, and other striking photographic effects.

[0040] While the invention has been disclosed with respect to a limited number of embodiments, numerous modifications and variations will be appreciated by those skilled in the art. It is intended that all such variations and modifications fall with in the scope of the following claims.

APPENDIX A

```
static void
radius_to_npasses_blur (float radius, int passesPerStep, int *pnpasses,
int *plastpassnpasses, float *pinterp)
{
int npasses, pass, lastpassnpasses;
float interp, variance, midvariance, lastvariance, passvariance, delta;

variance = radius * radius;
pass = 0;
lastvariance = 1.0f / 12.0f; /* variance for unit convolution */
delta = 1.0f;

/* compute next pass' variance (for pass 1) */
passvariance = lastvariance + delta;

while (variance > passvariance)
{
pass++;
lastvariance = passvariance;
delta = delta * 4.0f;
passvariance = lastvariance + delta;
}

/* use small (121) passes at the end to increase accuracy */
midvariance = (lastvariance + passvariance) * .5f;
lastpassnpasses = (variance <= midvariance) ? 1 : 2;

if (pass == 0)
{
npasses = passesPerStep;
if (lastpassnpasses == 1)
interp = variance / midvariance;
else
interp = (variance - midvariance) / (passvariance - midvariance);
}
else
{
npasses = (pass + 1) * passesPerStep;
if (lastpassnpasses == 1)
interp = (variance - lastvariance) / (midvariance - lastvariance);
else
interp = (variance - midvariance) / (passvariance - midvariance);
}

*pnpasses = npasses;
*plastpassnpasses = lastpassnpasses;
*pinterp = interp;
}
```